Per Container gibt es lokale Typvereinbarungen, die die jeweiligen Spezifika des Containers "kennen", nach außen sich jedoch gleich als abstrakte Konzepte verhalten:

```
namespace std {
        template <class T> // (fast) so aus <vector>
        class vector { public:
                typedef T                       value_type;
                typedef value_type*             pointer;
                typedef const value_type*       const_pointer;
                typedef value_type*             iterator;
                typedef const value_type*       const_iterator;
                typedef value_type&             reference;
                typedef const value_type&       const_reference;
                typedef size_t                  size_type;
                typedef ptrdiff_t               difference_type;
        .....};
} // end namespace std
```

Benutzung muss die konkreten Typen (der Elemente und des Containers!) NICHT kennen:

```
typedef vector<int> Ivec;
Ivec v;
// fill v ....
// iterate:

for (Ivec::iterator i = v.begin(); i != v.end(); ++i)
       do_something_with( *i );
...
// ist 42 drin:
Ivec::iterator where = find (v.begin(), v.end(), 42);
if (where != v.end())       // ja !
else                        // nein !
```

Anfang !

Ende !

Benutzung muss die konkreten Typen (der Elemente und des Containers!) NICHT kennen:

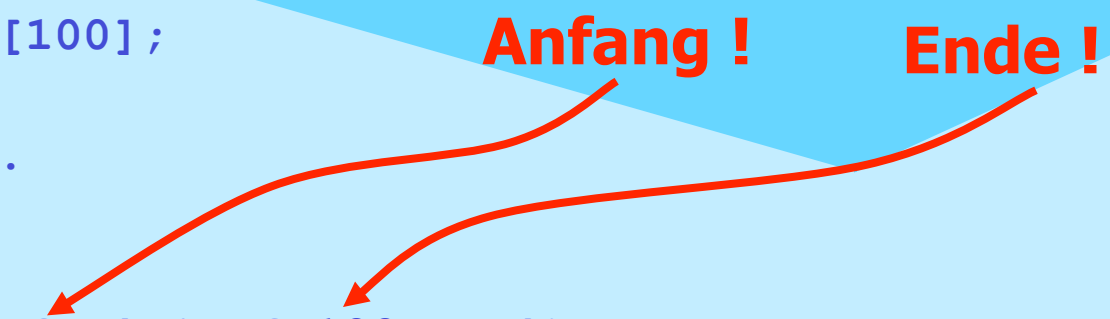
```
typedef set<int> Iset;
Iset s;
// fill s ....
// iterate:

for (Iset::iterator i = s.begin(); i != s.end(); ++i)
       do_something_with( *i );
...
// ist 42 drin:
Iset::iterator where = find (s.begin(), s.end(), 42);
if (where != s.end())       // ja !
else                        // nein !
```

Anfang !

Ende !

Benutzung muss die konkreten Typen (der Elemente und des Containers!) NICHT kennen, auch C-Felder können (als triviale Container) mit ihren Iteratoren (Zeiger) verwendet werden




```
typedef int F[100];
F f;
// fill f ....
// iterate:

for (int* i = f; i != f+100; ++i)
       do_something_with( *i );
...
// ist 42 drin:
int* where = find (f, f+100, 42);
if (where != f+100)          // ja !
else                         // nein !
```

Generische Algorithmen `(#include <algorithm>)`

von speziellen Containern durch Iteratoren entkoppelt!
standardisiert ist das Verhalten (ISO 14882/1998)

25.1.1 - For each [lib.alg.foreach]

```
template<class InputIterator, class Function>
  Function for_each(InputIterator first, InputIterator last, Function f);
```

-1- Effects: Applies f to the result of dereferencing every iterator in the range [first, last), starting from first and proceeding to last - 1.

-2- Returns: f.

-3- Complexity: Applies f exactly last - first times.

-4- Notes: If f returns a result, the result is ignored.

nicht die Implementation:

```
// bcc32:
template <class InputIterator, class Function>
Function for_each (InputIterator first, InputIterator last, Function f) {
    while (first != last) f(*first++);
    return f;
}
// g++ 3.2.2: siehe http://www.boost.org/libs/concept_check/concept_check.htm
template<typename _InputIter, typename _Function>
_Function for_each(_InputIter __first, _InputIter __last, _Function __f) {
    // concept requirements
    __glibcpp_function_requires(_InputIteratorConcept<_InputIter>)
    for ( ; __first != __last; ++__first)
      __f(*__first);
    return __f;
}
// vc++ 6.0: siehe http://www.dinkumware.com
template<class _II, class _Fn> inline
          _Fn for_each(_II _F, _II _L, _Fn _Op)
          {for (; _F != _L; ++_F)
                    _Op(*_F);
          return (_Op); }
```

Nicht-Modifizierende Algorithmen (#include <algorithm>)

`for_each`	eine (lesende) Operation auf alle Elemente anwenden
`find`	erstes Auftreten eines Wertes ermitteln
`find_if`	erste Erfüllung eines Prädikates ermitteln
`search`	erstes Auftreten einer Teilfolge ermitteln, gleiche Werte / Prädikat **! overloaded**
`find_end`	letztes Auftreten einer Teilfolge ermitteln, gleiche Werte
`find_end_if`	letztes Auftreten einer Teilfolge ermitteln, Prädikat
`find_first_of`	erstes Auftreten eines Elementes aus einem Bereich ermitteln, gleiche Werte
`find_first_of_if`	erstes Auftreten eines Elementes aus einem Bereich ermitteln, Prädikat
`adjacent_find`	erstes Auftreten benachbarter gleicher Elemente ermitteln, gleiche Werte / Prädikat **! overloaded**
`min_element`	Position des kleinsten Elements ermitteln, < / binäres bool Prädikat
`max_element`	Position des größten Elements ermitteln, < / binäres bool Prädikat
`count`	Elemente zählen, gleich Vorgabewert
`count_if`	Elemente zählen, für die Prädikat erfüllt
`equal`	Vergleich Bereich gegen Bereichsanfang, gleiche Werte / Prädikat
`lexicographical_compare`	Vergleich zweier Bereiche, elementweise gleich / Prädikat
`mismatch`	Position der ersten Abweichung bei Vergleich Bereich gegen Bereichsanfang gleiche Werte / Prädikat (Ergebnis: Iteratorpaar!)

Modifizierende Algorithmen (---> nicht für [multi]set/map !) (#include <algorithm>)

`copy`	einen Bereich an einen Bereichsanfang kopieren (Platz muss ausreichen!)
`copy_backward`	einen Bereich vor ein Bereichsende kopieren (Platz muss ausreichen!)
`swap_ranges`	Austausch Bereich gegen Bereichsanfang
`transform`	eine lesende und schreibende Operation auf alle Elemente anwenden auch mit Bereich und Bereichsanfang, die binär verknüpft werden und Ergebnisse an einen weiteren Bereichsanfang geschrieben werden (Platz muss ausreichen!)
`fill`	einen Bereich mit einem Wert füllen
`fill_n`	ab Bereichsanfang n Elemente mit einem Wert füllen (Platz muss ausreichen!)
`generate`	einen Bereich mit dem Ergebnis einer Operation füllen
`generate_n`	ab Bereichsanfang n Elemente mit dem Erg. einer Operation füllen (Platz muss ausreichen!)
`replace`	in einem Bereich alle Elemente mit altem Wert durch neuen Wert ersetzen
`replace_if`	in einem Bereich alle Elemente für die eine Prädikat gilt durch neuen Wert ersetzen
`replace_copy`	in einem Bereich alle Elemente mit altem Wert durch neuen Wert ersetzt an einen Bereichsanfang kopieren
`replace_copy_if`	in einem Bereich alle Elemente für die eine Prädikat gilt durch neuen Wert ersetzt an einen Bereichsanfang kopieren

Löschende Algorithmen (#include <algorithm>)

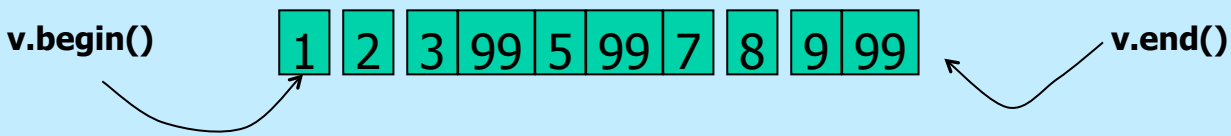
ACHTUNG: Größe und Ende des manipulierten Containers ändert sich NICHT!
Ergebnis ist immer das neue Ende des Containers!

`remove`	in einem Bereich alle Elemente löschen, die gleich einem Wert sind
`remove_if`	in einem Bereich alle Elemente löschen, für die ein Prädikat gilt
`remove_copy`	aus einem Bereich alle Elemente die gleich einem Wert sind, gelöscht an einen Bereichsanfang kopieren
`remove_copy_if`	aus einem Bereich alle Elemente für die ein Prädikat gilt, gelöscht an einen Bereichsanfang kopieren
`unique`	in einem Bereich gleiche aufeinanderfolgende Elemente zu einem kollabieren **UND** in einem Bereich bzgl. einem Prädikat gleiche aufeinanderfolgende Elemente zu einem kollabieren
`unique_copy`	aus einem Bereich gleiche aufeinanderfolgende Elemente zu einem kollabiert an einen Bereichsanfang kopieren **UND** aus einem Bereich bzgl. einem Prädikat gleiche aufeinanderfolgende Elemente zu einem kollabiert an einen Bereichsanfang kopieren

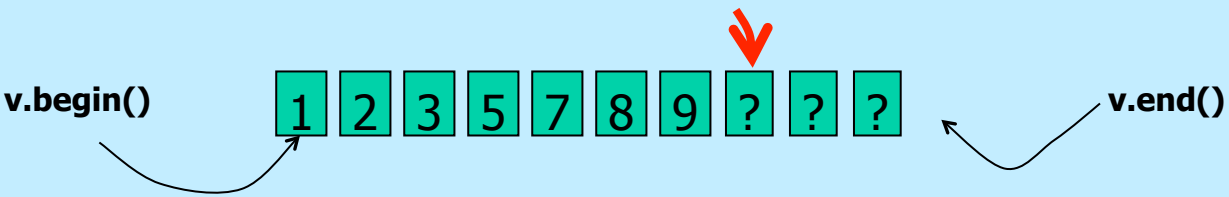
Löschende Algorithmen

ACHTUNG: Größe und Ende des manipulierten Containers ändert sich NICHT! Ergebnis ist immer das neue Ende des Containers!

Scott Meyers: Effective STL Item 32 (pp.139): "Follow `remove`-like Algorithms by `erase` if you really want to remove something."




```
vector<int>::iterator newEnd(remove(v.begin(), v.end(), 99));
```




```
v.erase((remove(v.begin(), v.end(), 99), v.end()); // !!!
```

Löschende Algorithmen

ACHTUNG: Auch wenn damit die logische Konsistenz wiederhergestellt ist, wird bei Vektoren kein Speicheplatz zurückgegeben!

Szenario: Sammle Kandidaten K in einem `vector<K> k` (~100.000) treffe Auswahl (die besten 10) entferne und lösche den Rest: der Vektor belegt immer noch 100.000 Elemente im Speicher :-(

Scott Meyers: Effective STL Item 17 (pp.77): "Use » the `swap` trick« to trim excess capacity."

```
vector<K>(k).swap(k); // wie bitte ???
```

```
// {
//     vector<K> tmp(k); // copy-ctor erzeugt so viele elemente wie nötig
//     tmp.swap(k);// k ist der neue (reduzierte) Vektor, tmp wird freigegeben
//  }
```

Mutierende Algorithmen (#include <algorithm>)

`reverse`	in einem Bereich die Reihenfolge aller Elemente umkehren
`reverse_copy`	aus einem Bereich alle Elemente in umgekehrter Reihenfolge an einen Bereichsanfang kopieren
`rotate`	einen Bereich so rotieren, dass ein neuer Bereichsanfang zum ersten Element wird
`rotate_copy`	einen Bereich rotiert, an neuem Bereichsanfang an einen Bereichsanfang kopieren
`next_permutation`	in einem Bereich die nächste Permutation erzeugen, liefert false, wenn letzte (lexikogr. aufsteigende) erreicht ist, sonst true
`prev_permutation`	in einem Bereich die vorhergehende Permutation erzeugen, liefert false, wenn letzte (lexikogr. absteigende) erreicht ist, sonst true
`random_shuffle`	einen Bereich zufällig nach einem impliziten oder expliziten Zufallsgenerator 'verwürfeln'
`partition`	in einem Bereich alle Elemente, für die ein Prädikat gilt, nach vorn schieben, Ergebnis ist die erste Position, an der das Prädikat nicht gilt
`stable_partition`	in einem Bereich alle Elemente, für die ein Prädikat gilt, nach vorn schieben, Ergebnis ist die erste Position, an der das Prädikat nicht gilt, Teilmengen bleibt erhalten

Reihenfolgen in

Sortierende Algorithmen (#include <algorithm>)

`sort`	einen Bereich nach < oder einer Relation sortieren
`stable_sort`	einen Bereich nach < oder einer Relation sortieren, gleiche Elemente bleiben in relativer Position zueinander !
`partial_sort`	einen Bereich bis zu einer Position nach < oder einer Relation sortieren
`partial_sort_copy`	aus einem Bereich bis zu einer Position nach < oder einer Relation sortiert in einen anderen Bereich kopieren
`nth_element`	einen Bereich nach < oder einer Relation um eine Position sortieren, so dass diese an der richtigen Stelle steht (links <=, rechts >)
`make_heap`	einen Bereich nach < oder einer Relation in einen Heap umwandeln (∀ i : h[i]>h[2*i+1] und h[i]>h[2*i+2])
`push_heap` [f,l)	das letzte Element eines Bereichs [f, l) in einen Heap [f, l-1) zu einem Heap einbauen (make_heap, push_back, push_heap) (nach < oder einer Relation)
`pop_heap`	das erste (nach < oder einer Relation größte) Element mit dem letzten vertauschen und einen neuen Heap in [f, l-1) erzeugen
`sort_heap`	einen Heap nach < oder einer Relation sortieren (Ergebnis ist bei < kein Heap mehr) O(N*log(N))

Algorithmen für sortierte Bereiche (#include <algorithm>)

`lower_bound`	in einem (nach < oder einer Relation) sortierten Bereich die erste Position ermitteln, an der ein Wert sortiert nach < oder einer Relation eingefügt werden kann
`upper_bound`	ein einem (nach < oder einer Relation) sortierten Bereich die letzte Position ermitteln an der ein Wert sortiert nach < oder einer Relation eingefügt werden kann
`equal_range`	in einem nach < oder einer Relation sortierten Bereich die erste und letzte Position für sortiertes Einfügen als `pair<FwdIter, FwdIter>` ermitteln
`binary_search`	ist ein Wert in einem nach < oder einer Relation sortierten Bereich enthalten ?
`includes`	ist ein nach < oder einer Relation sortierter Bereich in einem anderen nach < oder einer Relation sortierten Bereich enthalten ?

Algorithmen für sortierte Bereiche (#include <algorithm>)

merge zwei nach < oder einer Relation sortierte Bereiche an einen Bereichsanfang sortiert mischen (Platz muss ausreichen!)

inplace_merge zwei nach < oder einer Relation sortierte Teilbereiche die direkt hintereinander liegen werden sortiert gemischt (f1, l1f2, l2)

set_union zwei nach < oder einer Relation sortierte Bereiche an einen Bereichsanfang sortiert mischen, Elemente, die in beiden vorkommen werden nur einmal erfasst

set_intersection aus zwei nach < oder einer Relation sortierten Bereichen wird die Schnittmenge an einen Bereichsanfang übertragen

set_difference aus zwei nach < oder einer Relation sortierten Bereichen wird die Differenzmenge (im ersten und nicht im zweiten) an einen Bereichsanfang übertragen

set_symmetric_difference aus zwei nach < oder einer Relation sortierten Bereichen wird die Komplementärmenge (im ersten und nicht im zweiten oder umgekehrt) an einen Bereichsanfang übertragen

Numerische Algorithmen (#include <numeric>)

`accumulate`	bildet die Summe (bzw. Anwendung eines Operators) eines Initialwertes mit allen Elementen eines Bereichs
`partial_sum`	bildet die Summe (bzw. Anwendung eines Operators) aller Anfangsstücke eines Bereichs und kopiert diese ab einem Bereichsanfang
`adjacent_difference`	bildet die Differenzen (bzw. Anwendung eines Operators) jeweils benachbarter Elemente eines Bereichs und kopiert diese ab einem Bereichsanfang (erstes Element wird übernommen)
`inner_product`	bildet die Summe (bzw. Anwendung eines Operators) aus einem Initialwert und dem Skalarprodukt (bzw. Anwendung eines weiteren Operators) eines Bereichs und eines Bereichsanfangs

Folgende Containertypen existieren:

Typ	Header	Charakteristik	Iteratoren
`vector`	`<vector>`	dynamische Felder mit wahlfreiem Zugriff, erweiterbar am Ende	`RandomAccess`
`deque`	`<deque>`	dynamische Felder mit wahlfreiem Zugriff, erweiterbar am Anfang und am Ende	`RandomAccess`
`list`	`<list>`	doppelt verkette Listen ohne wahlfreiem Zugriff, nicht sortiert	`Bidirectional`
`set`	`<set>`	Mengen mit impliziter Sortierung, keine Duplikate	`Bidirectional`
`multiset`	`<set>`	Mengen mit impliziter Sortierung, Duplikate erlaubt	`Bidirectional`
`map`	`<map>`	Mengen von Schlüssel/Wert- Paaren mit impliziter Sortierung nach dem Schlüssel, keine Duplikate	`Bidirectional`
`multimap`	`<map>`	Mengen von Schlüssel/Wert- Paaren mit impliziter Sortierung nach dem Schlüssel, Duplikate erlaubt	`Bidirectional`

gemeinsame (generische) Operationen aller Container-Klassen:

1. Erzeugung und Zerstörung

Ausdruck	Bedeutung
`Container<Typ> m`	erzeugt einen leeren Container ohne Elemente
`Container<Typ> m1 (m2)`	erzeugt einen Container als Kopie eines anderen (gleichen Typs)
`Container<Typ> m (f, l)`	erzeugt einen Container und initialisiert ihn mit Kopien der Elemente aus dem Bereich [f, l)
`m.~Container<Typ> ()`	löscht alle Elemente und gibt deren Speicherplatz frei

gemeinsame (generische) Operationen aller Container-Klassen:

2. Nichtverändernde Operationen

Ausdruck	Bedeutung
`m.size ()`	aktuelle Anzahl von Elementen
`m.empty ()`	leer? (entspricht m.size()==0 ABER SCHNELLER!)
`m.max_size ()`	maximal mögliche Anzahl von Elementen
`m1 == m2` `m1 != m2`	Gleichheit (== auf alle Elemente angewendet) Ungleichheit (dito)

gemeinsame (generische) Operationen aller Container-Klassen:

3. Zuweisende Operationen

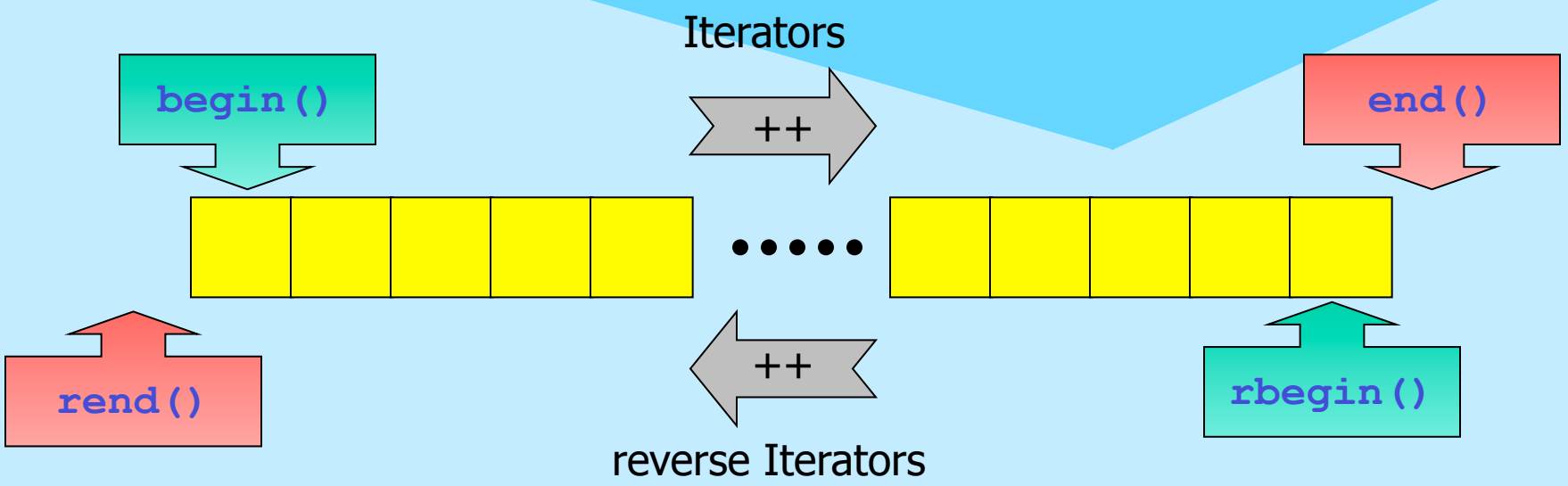
Ausdruck	Bedeutung
`m1 = m2`	Zuweisung
`m1.swap (m2)`	Vertauschen aller Elemente
`swap (m1, m2)`	!! immer schneller als m1 = m2

gemeinsame (generische) Operationen aller Container-Klassen:

4. Zugreifende Operationen

Ausdruck	Bedeutung
`m.begin ()`	Iterator auf das erste Element
`m.end ()`	Iterator hinter das letzte Element
`m.rbegin ()`	Iterator auf das letzte Element
`m.rend ()`	Iterator vor das erste Element

Abstrakte Iteratoren



alle Operationen liefern sog. const iterators für konstante Container
(über solche ist der Container nicht veränderbar)

gemeinsame (generische) Operationen aller Container-Klassen:

5. Einfügende/Löschende Operationen

Ausdruck	Bedeutung
`m.insert (pos, e)`	Kopie von e bei pos einfügen (Rückgabewert und Bedeutung von pos hängen von Containertyp ab)
`m.erase (pos)`	löscht Element an der Position pos
`m.erase (f, l)`	löscht alle Elemente aus dem Bereich [f, l)
`m.clear ()`	leert den Container

Wichtige Eigenschaften der Container

- die Effizienz aller Operationen ist optimiert, der Standard macht Vorgaben zu oberen Grenzen
- die verfügbaren Implementationen orientieren sich am state of the art in ihrer algorithmischen Umsetzung
- die generischen (container-unabhängigen) Algorithmen sind in `<algorithm>` definiert
- häufig gibt es für spezielle Container bessere Algorithmen als Memberfunktionen (z.B. `set<T>::find`)
- nicht alle Kombinationen von Algorithmen und Containern sind möglich (sinnvoll) (z.B. `sort` auf `set`: Mengen sind bereits sortiert, `sort` auf `list` ist nicht möglich)
- Alle Container arbeiten mit einer Wertesemantik, d.h. Elemente werden immer kopiert !

Timer.h

```
#include <ctime>
#include <iostream>
const double millis= 1000000.0;
class Timer { long t_;
    void report()
    { std::cout<<t_/millis<<"s"<<std::endl; }
public:
    Timer(): t_(clock()){}
    ~Timer(){ t_=clock()-t_;report();}
};
```

Member-Algorithmen sind besser als globale:

```
#include <set>
#include <iostream>
#include <algorithm>
#include "Timer.h"
using namespace std;
typedef set<int> Set;

main() {
  Set s;  int probe = rand();
  {
        cout<<"Generating 1.000.000 nodes: "<<flush;
        Timer t;
        for(int i=0; i<500000; ++i) s.insert(rand());
        s.insert(probe);
        for(int i=1; i<500000; ++i) s.insert(rand());
        cout<<s.size()<<" Elemente in der Menge"<<endl;
  }
```

Member-Algorithmen sind besser als globale:

```
{
        cout<<"s.find(...): "<<flush;
        Timer t;
        for (int i=0; i<1000; ++i) s.find(probe); // member
}
{
        cout<<"find (...): "<<flush;
        Timer t;
        for (int i=0; i<1000; ++i)
                find(s.begin(), s.end(), probe); // generic
}
}
```

```
Generating 1.000.000 nodes:
999752 Elemente in der Menge
5.22s
s.find(...): 0s
find (...): 346.93s !!!
```

nicht alle Kombinationen von Algorithmen und Containern sind möglich:

```
// vector, algorithm, iostream
using namespace std;
typedef vector<int> V;

template <class CT>
void out(const CT& c) {
  for(typename CT::const_iterator i=c.begin(); i!=c.end(); ++i)
    std::cout<<*i<<' ';
  std::cout<<std::endl;
}

int main() {
  V v;
  for(int i=0; i<10; ++i) v.push_back(rand());
  out(v);
  sort(v.begin(), v.end());
  out(v);
}
```

nicht alle Kombinationen von Algorithmen und Containern sind möglich: `ll.c`

```
// list, algorithm, iostream
using namespace std;
typedef list<int> L;

template <class CT>
void out(const CT& c) {
  for(typename CT::const_iterator i=c.begin(); i!=c.end(); ++i)
    std::cout<<*i<<' ';
  std::cout<<std::endl;
}

int main() {
  L l;
  for(int i=0; i<10; ++i) l.push_back(rand());
  out(l);
  sort(l.begin(), l.end()); // ??? list hat keine random access iteratoren!
  out(l);
}
```

```
l.sort(); // OK
```

nicht alle Kombinationen von Algorithmen und Containern sind möglich:

```
mio ahrens 72 ( c++/experimente ) > make ll
/usr/include/g++/stl_algo.h: In function `void sort<_List_iterator<int,int
&,int
 *> >(_List_iterator<int,int &,int *>, _List_iterator<int,int &,int *>)':
ll.cc:17:   instantiated from here
/usr/include/g++/stl_algo.h:1320: no match for `_List_iterator<int,int &,int
*>
& - _List_iterator<int,int &,int *> &'
.....
mio ahrens 73 ( c++/experimente ) > ~/STLfilt/gfilt ll.cc
BD Software STL Message Decryptor v2.31 for gcc (03/03/2003)
stl_algo.h: In function
   `void sort<list<int>::iter>(list<int>::iter, list<int>::iter)':
ll.cc:17:   instantiated from here
stl_algo.h:1320: no match for `list<int>::iter & - list<int>::iter &'
    [STL Decryptor: Suppressed 14 more STL standard header messages]
```

Vektoren (`#include <vector>`)

- dynamische Arrays eines beliebigen Typs mit wahlfreiem Zugriff
- Abstraktion von C-Feldern, unsortiert
- alle Algorithmen sind anwendbar (RandomAccessIterator)
- sehr gutes Zeitverhalten beim Löschen und Einfügen am Ende
- ansonsten ist jedes Löschen/Einfügen mit dem Verschieben (Zuweisen) von Elementen verbunden ! (--> schlechtes Zeitverhalten)
- Vektoren wachsen dynamisch:

`size()`	Anzahl der aktuell enthaltenen Elemente
`max_size()` (impl.abh.)	Anzahl der maximal möglichen Elemente
`capacity()`	Anzahl der insgesamt (ohne) Reallokierung aufnehmbaren Elemente
`reserve(size_type)`	Freihalten von Platz

Vektoren (#include <vector>)

- beim Einfügen/Löschen werden u.U. Iteratoren ungültig, bei Reallokierung = Copykonstruktor + Destruktor am alten Platz / Element !!! werden alle Iteratoren ungültig

Ausdruck	Bedeutung
`vector<E> m (n)`	n E-Elemente per default-Konstruktor
`vector<Elem> m (n, e)`	n E-Elemente als Kopie von e
`m.capacity ()`	freier Platz ohne Reallokierung
`m.reserve (size_type)`	Platz freihalten

Ausdruck	Bedeutung
`m.assign (n, e)`	Zuweisung von n Elementen, die als Kopie von e erzeugt werden
`m.assign (f, l)`	Zuweisung von Kopien der Elemente aus dem Bereich [f, l)
`m.at (n)`	Element am Index n (mit Prüfung ob es existiert, ggf. out_of_range Exc. !)
`m[n]`	Element am Index n (ohne Prüfung ob es existiert !)
`f.front ()`	das erste Element (ohne Prüfung ob es existiert!)
`f.back ()`	das letzte Element (ohne Prüfung ob es existiert!)

Ausdruck	Bedeutung
`m.insert (pos,e)`	fügt Kopie von e an pos ein, liefert Position des neuen Elements
`m.insert (pos, f, l)`	bei pos Kopien von [f, l) einfügen Resultat void (member template !)
`m.push_back (e)`	Kopie von e hinten anhängen
`m.pop_back ()`	löscht letztes Element (gibt nichts zurück)
`m.resize (n)`	Größe auf n ändern und ggf. mit dc Elementen auffüllen
`m.resize (n, e)`	Größe auf n ändern und ggf. mit Kopien von e auffüllen

dc - default constructed